Coordination Implications of Software Coupling in Open Source Projects

Chintan Amrit[1] and Jos van Hillegersberg[1],

[1]IS&CM Department, University of Twente,
PO Box 217
7500 AE Enschede, The Netherlands
{c.amrit, j.vanhillegersberg}@utwente.nl

Abstract. The effect of software coupling on the quality of software has been studied quite widely since the seminal paper on software modularity by Parnas [1]. However, the effect of the increase in software coupling on the coordination of the developers has not been researched as much. In commercial software development environments there normally are coordination mechanisms in place to manage the coordination requirements due to software dependencies. But, in the case of Open Source software such coordination mechanisms are harder to implement, as the developers tend to rely solely on electronic means of communication. Hence, an understanding of the changing coordination requirements is essential to the management of an Open Source project. In this paper we study the effect of changes in software coupling on the coordination requirements in a case study of a popular Open Source project called JBoss.

Keywords: Software Coupling, Propagation Cost, Clustered Cost, Open Source, Coordination

1 Introduction

Open Source developers generally rely on electronic means of communication, coordination in Open Source environments is difficult to achieve when compared to commercial software development. It is therefore essential for an Open Source project Manager to understand the changing coordination requirements in Open Source software in order to ensure successful coordination. While the coordination implication of software coupling has been suggested by various researchers [2-5], there has been little research done on the effect of the change in coupling on the coordination requirements of developers. Such research is especially important in the Open Source context, where the distributed and generally ad-hoc nature of development makes coordination of the development challenging.

MacCormack et al. [6] compare the architectures of Linux and Mozilla by comparing the pattern of distribution of their software coupling. They find that Linux had a more modular structure than the first version of Mozilla. While after a redesign the resulting architecture, Mozilla became more modular than the previous versions and even more modular than Linux. This result is in line with the view that in order to

have a successfully coordinated Open Source project one needs to have a loosely coupled and modular software [7]. Authors like O'Reilly [8] have claimed that Open Source software is inherently more modular than commercial software. Other authors have reasoned that Open Source software needs to be more modular so that the development process can be coordinated easily [7]. Paulson et al. [9], compare the coupling of Open Source projects (Apache, Linux and GCC) with three closed source projects. They do so, by comparing the growing versus the changing rate for software (as a tighter coupling will require more changes with each additional feature). Their results indicate that Open Source projects need more changes when new features are added. Hence, suggesting tighter coupling in Open Source projects than previously assumed. Parnas [1] described modularisation as a task assignment while Conway[2] analysed the relation between product architecture and the organizational structure. Since then, Conway's law [10] has come to denote the homomorphism between the product architecture (or software coupling [3]) and the organizational structure (or the communication between the software developers [3]). As the Open Source project gets developed, the software code evolves [11], and as a result the coordination requirements change [3]. As mentioned earlier, there has been little research done on the effect that the variation of software coupling has on the coordination requirements of the software developers. In this paper we try and fill this gap by analysing the effect of the changes in software coupling on the coordination requirements of the developers. We postulate that, if there is a sudden increase in the coupling of an Open Source system, then the coordination requirement among the developers' increases. Unless this coordination requirement is handled through communication, it could result in a coordination problem [12]. By conducting a case study of the of the JBoss application server, we observe the effect of the changes in coupling on the coordination of the project. The unique contribution of this paper lies in discussing the coordination implications of an increase in software coupling and then in demonstrating it through a case study that uses quantitative along with qualitative methods.

The rest of the paper is structured as follows; section 2 describes the Design Structure Matrices briefly along with the Clustered and Propagation Cost metrics used in this paper, section 3 describes the case study of JBoss, section 4 discusses the findings and finally section 5 concludes the paper.

2 Design Structure Matrix and Cost Metrics

In this section we describe the data structure and the metrics we use to study software coupling. Dependency Structure Matrices (DSM) have been used in engineering literature to represent the dependency between tasks, since the concept of the Design Structure Matrix was first proposed by Steward [13, 14]. A DSM highlights the inherent structure of a design by examining the dependencies between its component elements in a square matrix [13, 15]. Morelli et al. [16] describe a method to predict and measure coordination-type of communication within a product development organization. They compare predicted and actual communications in order to learn, to what extent an organizations communication patterns can be anticipated.

Sosa et al.[4] find a "strong tendency for design interactions and team interactions to be aligned," and show instances of misalignment are more likely to occur across organizational and system boundaries. Sullivan et al. [17] use DSMs to formally model (and value) the concept of information hiding, the principle proposed by Parnas to divide designs into modules [1]. Cataldo et al.[3] show how DSMs can be used to predict coordination in a software development organization and then they compare the predicted coordination DSM with the actual communication DSM. Sosa [5] builds on the DSM based method of Cataldo et al. [3] and provides a structured approach to identify the employees who need to interact and the software product interfaces they need to interact about. Amrit et al. [12, 18] take a similar approach and use DSMs to detect coordination problems in a software development environments.

We use the Software Dependency Matrix (the DSM of software dependencies) to calculate the Propagation Cost and Clustered Cost similar to what MacCormack et al. [6] do. Our unit of analysis is the source code file and we consider the function call dependencies among the files.

While the Propagation Cost assumes that the cost of dependencies between two elements are the same irrespective of where the elements lie (the path length between them), Clustered Cost assumes that the cost of dependency depends on whether the elements lie in the same cluster [6]. Together the Propagation and Clustered Cost measure both the number as well as the pattern of the software dependency [6]. In order to calculate the Propagation Cost, MacCormack et al. first raise their dependency matrix to successive powers of n and obtain the direct and indirect dependencies for successive path lengths [6]. They then obtain a Visibility Matrix by summing up all the successive powers of the dependency matrix. From the Visibility Matrix they calculate the "fan-in" and "fan-out" visibilities by summing along the columns or the rows and dividing the result with the total number of elements. As we consider undirected dependencies, we find the "fan-in" visibility to equal the "fan-out" visibility. The Propagation Cost measures the elements in the system that could be affected when a change is made to one element of the system (i.e. how the change *propagates*) [6].

Unlike the Propagation Cost, the Clustered Cost of an element depends on the location of the element (with respect to other elements). In order to measure the Clustered Cost, the DSM of the software call graph has to be first clustered. The clustering algorithm (described in [6]) tries to group all highly connected or dependent elements into one cluster. The clustering works by attaching a cost to each element, depending on where the element is located with respect to other elements (in the same vertical bus or in the same cluster)). The Clustered Cost of the software is then the summation of the individual Clustered Cost of the elements.

In the next section we describe the case study of the popular open source project JBoss. In the case study we describe how we apply the two metrics described in this section and the conclusions we draw from them.

3 Case Study of JBoss

JBoss project was started in 1999 by Marc Fleury who wanted to advance his research interests in middleware. JBoss Group LLC was incorporated in 2001 and JBoss became a corporation in 2004. After a few bids from big companies, JBoss was finally acquired by Red Hat in 2006. The JBoss Application Server is one of the main products of the JBoss project and is said to have pioneered the professional Open Source business model. JBoss has 79 listed developers and three project administrators of which one is the Chief Technical Officer (CTO) of JBoss.

The aim of the case study is to determine if there was a relation between the changes in the technical dependencies and the communication among the developers. For the technical dependencies, the JBoss Application Server (JBoss) source code was analysed over the period starting from May 2002 to December 2006 that covered the versions 3.0.0 to 4.0.3_sp1. We used a tool called TESNA [12] that uses DependencyFinder [19] to read the software code and create the DSMs. With the help of TESNA we could then calculate the Propagation and Clustered Cost based on the DSMs. The Lines of Code (KLOC) of the different versions of JBoss was also measured using the same tool.

To determine the communication patterns used by the developers, we analysed the Mailing List archive of JBoss. The JBoss Mailing List is used to discuss the development of the system, report bugs, coordinate the bug fixes, as well as discuss new features before and after the release of each version. An analysis of the different mediums of coordination in JBoss revealed that the Mailing List was the primary means of coordination. This is the case, as the usage of private means to communicate is considered unlikely, given the trend of openness in Open Source projects [20]. In order to find out the timeline around which developers discussed a particular release, we needed to first find out the coordination mechanisms used by the developers. We performed a qualitative analysis of the messages in the Mailing List archive where we read randomly selected mails (around each release) looking for coordination mechanisms as described in previous literature. The following post mailed on 28th of June shows how the management of each release was undertaken by one of the Project Leaders (Scott Stark in this case).

> Its about 36 hours until I'm planning on cutting the 3.0.1 release. Any changes you want in 3.0.1 should be in by Sat Jun 29 18:00:00 2002 GMT.
>
> xxxxxxxxxxxxxxxxxxxxxxxx
>
> Scott Stark

This post also shows that the planning for a release was done around a month earlier to the release, as the release date for version 3.0.1 was on 6th August 2002. While the following post shows another instance of a post reporting a fix for a bug.

> Sender: d_jencks
>
> Logged In: YES
>
> user_id=60525

> I believe I have fixed this in HEAD. I'd appreciate verification before I backport it to 3.2, since it is a substantial refactoring of the ejb deployment/service lifecycle code. I'll close this after backporting to 3.2.

This post shows two important mechanisms; (i) the request for verification implying the coordination mechanism of code review as was described by Rigby et al. [21], (ii) the one which d_jenks refers to as "backport". By "backport" the author refers to making changes to the previous version well after the release (2002-08-27). This coordination mechanism coincides with what was observed by Yamauchi et al. [22], namely, a bias towards action first and coordination later. Given that the planning for the release and the coordination for the bugs in the release was conducted around a month before and a month after the release respectively, we decided to consider the messages related to a release over a three month window. Hence, the Mailing Lists were analysed from one month before each release to one month after each release, corresponding to the period of analysis of the JBoss code (i.e. from April 2002 to January 2007). We decided to consider all the messages in the three months window, as messages dealing with the coordination of the community for the following reasons:

1) The threads containing more than one message is naturally a discussion thread implying coordination between messages
2) Threads containing only one message were mostly announcements such as "Build Fixed" that warrants no further replies. However, such posts are also coordination alerts for the community to not worry about the compilation part of the particular version and to concentrate on other work.

Figure 1 describes the variation of the Propagation Cost of JBoss over the different versions, while Figure 2 denotes the variation of the Clustered Cost of JBoss over different versions. In both figures and particularly in Figure 1 we notice a sharp rise in the Clustered Cost for version 3.2.7. While the increase in the Propagation Cost is minor, the increase in the Clustered Cost for version 3.2.7 is quite marked. We calculated the KLOC (Lines of Code in thousands) of each of the versions to see how much code was actually added. Figure 3 shows the variation of KLOC over the different versions of JBoss. As can be seen from the figure, the trend is similar to the variation of coupling seen in Figures 1 and 2. The largest increase in KLOC, as evident from the slope of the graph in Figure 3, occurs for version 3.2.7. Clearly showing that for version 3.2.7 not only has the complexity of the code increased (with the increased coupling), but also the size.

This increase in modularity of the project causes an increase in the coordination requirement [3] and therefore require an increased amount of coordination to resolve the extra dependencies and features included for version 3.2.7.

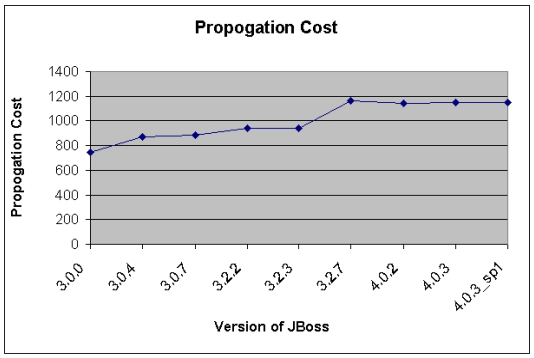

Figure 1: The variation of Propagation Cost of JBoss over different versions

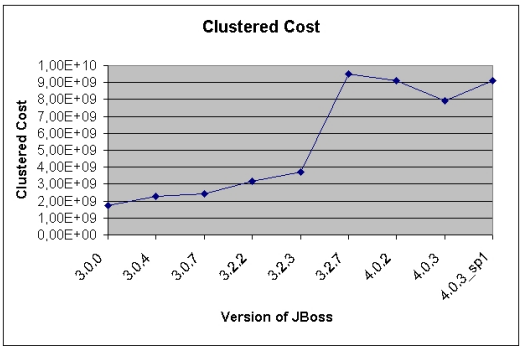

Figure 2: The variation of Clustered Cost of JBoss over different version

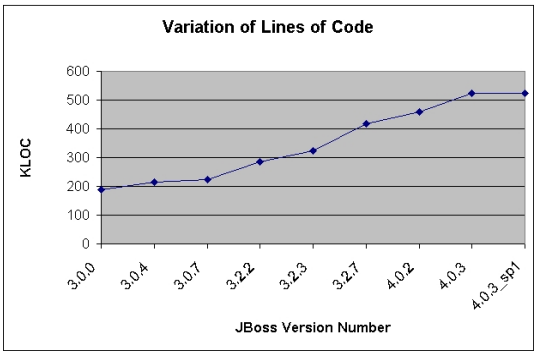

Figure 3: Variation of KLOC with Version number of JBoss

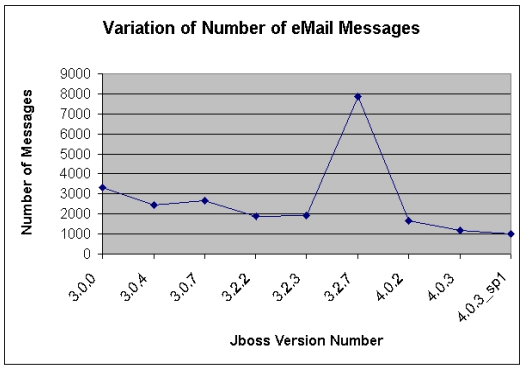

Figure 4: Variation of the Number of eMail messages with JBoss Version number

Figure 4 describes the variation in the number of messages over the different versions of JBoss. We see a large increase in the number of messages for discussing the features and bugs for version 3.2.7. The increase in the number of messages is nearly 5000 and twice as much as the average number of messages (2650) discussing other versions.

4 Discussion and Conclusion

Though one needs to analyse the mails more closely to ascertain if they are indeed discussing the particular version, one can say with some confidence that this sharp increase in messages can be explained by the increased need for coordination. This

increased need for coordination arises from the increased number of couplings and related features of JBoss in the release. Such an increase in the communication of the developers in the eMail List can indicate how the developers of JBoss satisfy the changing coordination needs for different versions and as a result remains a successful Open Source project. Had the coordination not increased to offset the increase in coupling and complexity of the software, we might have noticed a coordination problem as described by deSouza [23] and Amrit et. al [12].

In this paper we addressed the implications of coordination of an Open source project when the software coupling in the project changes. Clearly, the change in software coupling causes a change in the coordination requirements of the project as suggested by [2, 3, 6]. Unless this increase in the coordination requirement is compensated by an increase in communication related to the coordination, (as in the JBoss case study) one can expect consequences to the software quality of the project. Hence, this research has implications for the Open Source project manager. As such a manager has to be aware of the increased coordination requirement arising from changes in the project's software coupling.

The contribution of the research in this paper is twofold; (i) a discussion on the coordination implications of an increase in software coupling and (ii) the case study demonstrating the coordination implication using appropriate metrics like Propagation, Clustered Cost, KLOC and number of Mailing List messages. The email archive of JBoss also reveals two particular coordination mechanisms used to coordinate the development of JBoss, namely code reviews [21] and post-release coordination [22]. Future work can look at why the clustered and propagation cost differed in describing the coordination requirements in this case. Also, future work could look into different perspectives of comparing the effect of other technical dependencies on social coordination in Open Source projects. We are also studying the effect the change of coupling has on the health of the Open Source project.

References

1. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM, Vol. 15, New York, NY, USA (1972) 1053--1058
2. Conway, M.: How do Committees Invent. Datamation, Vol. 14 (1968) 28-31
3. Cataldo, M., Wagstrom, P., Herbsleb, J.D., Carley, K.M.: Identification of coordination requirements: implications for the Design of collaboration and awareness tools. Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work. ACM Press, Banff, Alberta, Canada (2006)
4. Sosa, M.E., Eppinger, S.D., Rowles, C.M.: The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. J Manage. Sci. **50** (2004) 1674-1689
5. Sosa, M.E.: A structured approach to predicting and managing technical interactions in software development. Research in Engineering Design **19** (2008) 47-70
6. MacCormack, A., Rusnak, J., Baldwin, C.Y.: Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Management Science **52** (2006) 1015-1030

7. Mockus, A., Fielding, R.O.Y.T., Herbsleb, J.D.: Two Case Studies of Open Source Software Development: Apache and Mozilla. ACM Transactions on Software Engineering and Methodology **11** (2002) 309-346
8. O'Reilly, T.: Lessons from open-source software development. Commun. ACM **42** (1999) 32-37
9. Paulson, J.W., Succi, G., Eberlein, A.: An Empirical Study of Open-Source and Closed-Source Software Products. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (2004) 246-256
10. Herbsleb, J., D., Grinter, R., E. : Architectures, Coordination, and Distance: Conway's Law and Beyond. IEEE Software, Vol. 16, Los Alamitos, CA, USA (1999a) 63--70
11. Koch, S.: Software evolution in open source projects—a large-scale investigation. Journal of Software Maintenance and Evolution: Research and Practice **19** (2007) 361-382
12. Amrit, C., van Hillegersberg, J.: Detecting Coordination Problems in Collaborative Software Development Environments. Information Systems Management **25** (2008) 57 - 70
13. Steward, D.: The design structure system: a method for managing the design of complex systems. IEEE Transactions on Engineering Management **28** (1981) 71-74
14. Steward, D.V.: Partitioning and tearing systems of equations. SIAM J. Numer. Anal **2** (1965) 345-365
15. Eppinger, S.D., Whitney, D.E., Smith, R.P., Gebala, D.A.: A model-based method for organizing tasks in product development. Research in Engineering Design **6** (1994) 1-13
16. Morelli, M.D., Eppinger, S.D., Gulati, R.K.: Predicting technical communication in product development organizations. Engineering Management, IEEE Transactions on **42** (1995) 215-222
17. Sullivan, K., J., Griswold, W., G., Cai, Y., Hallen, B.: The structure and value of modularity in software design. Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering. ACM Press, Vienna, Austria (2001) 99-108
18. Amrit, C., Van Hillegersberg, J.: Exploring the Impact of Socio-Technical Core-Periphery Structures in Open Source Software Development. Journal of Information Technology, forthcoming (2010)
19. Tessier, J.: Dependency Finder. (Retrieved on March 1st 2009)
20. Raymond, E.: The Cathedral and the Bazaar. Knowledge, Technology, and Policy **12** (1999) 23-49
21. Rigby, P.C., German, D.M., Storey, M.A.: Open source software peer review practices: a case study of the apache server. Proceedings of the 13th international conference on Software engineering (2008) 541-550
22. Yamauchi, Y., Yokozawa, M., Shinohara, T., Ishida, T.: Collaboration with Lean Media: how open-source software succeeds. Proceedings of the 2000 ACM conference on Computer supported cooperative work (2000) 329-338
23. de Souza, C., R. B., Redmiles, D., Cheng, L.-T., Millen, D., Patterson, J.: Sometimes you need to see through walls: a field study of application programming interfaces. CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work, New York, NY, USA (2004) 63--71